Efficient Multi-Class Probabilistic SVMs on GPUs

Zeyi Wen[†1], Jiashuai Shi[†‡2], Bingsheng He[†3], Jian Chen[‡4], Yawen Chen[‡5]
[†]National University of Singapore
[‡]South China University of Technology
[1,3]{wenzy,hebs}@comp.nus.edu.sg, [2,5]{shijiashuai,ywchenscut}@gmail.com, [4]ellachen@scut.edu.cn

Abstract—Recently, many researchers have been working on improving other traditional machine learning algorithms (besides deep learning) using high-performance hardware such as Graphics Processing Units (GPUs). The recent success of machine learning is not only due to more effective algorithms, but also more efficient systems and implementations. In this paper, we propose a novel and efficient solution to multi-class SVMs with probabilistic output (MP-SVMs) accelerated by GPUs. MP-SVMs are an important technique for many pattern recognition applications. However, MP-SVMs are very time-consuming to use, because using an MP-SVM classifier requires training many binary SVMs and performing probability estimation by combining results of all the binary SVMs. GPUs have much higher computation capability than CPUs and are potentially excellent hardware to accelerate MP-SVMs. Still, two key challenges for efficient GPU accelerations for MP-SVM are: (i) many kernel values are repeatedly computed as a binary SVM classifier is trained iteratively, resulting in repeated accesses to the high latency GPU memory; (ii) performing training or estimating probability in a highly parallel way requires a much larger memory footprint than the GPU memory.
To overcome the challenges, we propose a solution called *GMP-SVM* which exploits two-level (i.e., binary SVM level and MP-SVM level) optimization for training MP-SVMs and high parallelism for estimating probability. GMP-SVM reduces high latency memory accesses and memory consumption through batch processing, kernel value reusing and sharing, and support vector sharing. Experimental results show that GMP-SVM outperforms the GPU baseline by two to five times, and LibSVM with OpenMP by an order of magnitude. Also, GMP-SVM produces the same SVM classifier as LibSVM.

Index Terms—Machine Learning, Multi-class probabilistic SVMs, Graphics Processing Units.

1 Introduction

The development of more effective algorithms and of more efficient algorithms are the two key factors to the success of machine learning in the big data era. There is no doubt that high-performance hardware such as Graphics Processing Units plays an important role in the success. Recently, many researchers are rethinking about improving other traditional machine learning algorithms (besides deep learning) using the high-performance hardware [1], [2]. In this paper, we propose a novel and efficient solution to multi-class SVMs with probabilistic output (MP-SVMs) accelerated by GPUs. MP-SVMs have important applications in pattern recognition tasks such as medical image retrieval and image classification [3], [4], [5].

The most common implementation of MP-SVMs is through pairwise coupling (a.k.a. one-against-one), because the pairwise coupling method outperforms other methods [6]. LibSVM [7], an open source user friendly toolkit, is such implementation and has made the use of MP-SVM easier. However, a key barrier that hinders the wide use of MP-SVMs is its high training and probability estimation[1] cost, since an MP-SVM classifier consists of many binary SVMs. Those costs can be prohibitively high for ever increasingly large datasets. As we can see from Table 1, LibSVM without OpenMP takes about one hour on training an MP-SVM classifier for the MNIST dataset and another hour on estimating probability. For MNIST8M—a bigger dataset, LibSVM needs a few days to process. Even enabling OpenMP for LibSVM on a workstation with 20 CPU cores, handling MNIST8M still takes a long time (i.e., 22 hours on training). More details on this experimental setup can be found in Section 4. Encourged by the recent success of GPUs in machine learning, we investigate whether and how we can improve the performance of MP-SVM using GPU accelerations.

GPUs have much higher computation capability than CPUs and are potentially excellent hardware to accelerate MP-SVMs. A naive approach of using GPUs is to train the binary SVMs on the GPU one by one, and to estimate probability for multiple instances using one binary SVM at a time. The naive approach has a relatively small memory footprint which fits into the GPU. We call this approach the *GPU baseline*. Our experimental results (cf. Table 1) show that the GPU baseline is about three times faster in training and ten times faster in estimating probability than LibSVM with OpenMP. However, the GPU baseline is still inefficient (e.g., around ten hours in training for MNIST8M), because it severely underutilizes the GPU hardware. Two key challenges are: (i) many kernel values are repeatedly computed as a binary SVM classifier is trained iteratively, resulting in repeated accesses to the high latency GPU memory; (ii) performing training or estimating probability in a highly

1. "Probability estimation" and "prediction" are used interchangeably in this paper depending on which terminology is more natural in the context.

TABLE 1: Elapsed time (sec) comparison among LibSVM, and GPU based algorithms

Dataset	LibSVM without OpenMP		LibSVM with OpenMP		GPU baseline		our CPU version of GMP-SVM		our GMP-SVM	
	training	prediction	training	prediction	training	prediction	training	prediction	training	prediction
CIFAR-10	22,523	19,090	4,659	3,074	1,200.22	212.5	495.65	74.29	220.7	29.3
MNIST	3,847.96	3,113.16	429.1	245.7	100.83	30.46	96.55	12.04	34.1	4.62
MNIST8M	810,794	1.2x10^6	78,856	79,840	35,390	2,945.87	32,470.37	9,973.41	7,134.12	927.06

parallel way requires a much larger memory footprint than the GPU memory.

To address the challenges, we propose a highly efficient parallel solution called "*GMP-SVM*" which exploits two-level optimization for training MP-SVMs and high parallelism for estimating probability. GMP-SVM reduces high latency memory accesses and memory consumption through batch processing, kernel value reusing and sharing, and support vector sharing. In the binary SVM level, we compute kernel values in batches with the consideration of reusing kernel values via a GPU buffer. In the MP-SVM level, we develop techniques to concurrently train multiple binary SVMs with kernel value sharing among the binary SVMs. When estimating probability, GMP-SVM concurrently computes the probabilities for multiple instances using multiple binary SVMs with support vector and kernel value sharing among the SVMs. GMP-SVM can train MP-SVMs for MNIST in only 34 seconds and estimate probability almost instantly (cf. Table 1); in the MNIST8M dataset, GMP-SVM finishes training within two hours and estimating probability in 16 minutes. To investigate the significance of GPUs, we also compare GMP-SVM with our multi-threaded CPU version of GMP-SVM, the results show that GMP-SVM achieves about three times speedup over its CPU counterpart.

In summary, our key contributions in this paper are as follows. First, we design a highly efficient GPU solution (i.e., GMP-SVM) for training MP-SVMs and performing probability estimation. Our solution reduces repeated computation and addresses resource underutilization of the GPU baseline. GMP-SVM is integrated into the open-source project on GitHub at https://github.com/zeyiwen/thundersvm. The project has attracted 680+ stars and 80+ forks as of 24 Jun 2018. Second, we conduct comprehensive experiments to study the efficiency of GMP-SVM. Experimental results show that GMP-SVM outperforms the GPU baseline by two to five times, and LibSVM with OpenMP by an order of magnitude. Additionally, GMP-SVM produces the same SVMs as LibSVM. Note that, GMP-SVM is three to ten times faster than its CPU-based counterpart, which demonstrates the efficiency of our algorithmic design and the effectiveness of GPU accelerations.

The remainder of this paper is structured as follows. We present preliminaries on multi-class probabilistic SVMs in Section 2. Then we elaborate our implementation for multi-class probabilistic SVMs on GPUs in detail in Section 3. A comprehensive experimental study is provided in Section 4. After that, we discuss the related work in Section 5. Finally, we conclude the paper in Section 6.

2 PRELIMINARIES

In this section, we first present details of SVMs, *Sequential Minimal Optimization* (SMO) [8] for training SVMs, and converting the output of SVMs into probability. Then, we discuss MP-SVMs, and present key features of GPUs.

2.1 Support Vector Machines (SVMs)

Here, we discuss the basis form of SVMs [9] which are used for binary classification problems. This type of SVMs is also called binary SVMs which is the building block of MP-SVMs as we will see later in this section. Formally, an instance $\boldsymbol{x}_i$ is attached with an integer $y_i \in \{+1, -1\}$ as its label. A positive (negative) instance is an instance with the label of $+1$ (-1). Given a set $\mathcal{X}$ of n training instances, the goal of the SVM training is to find a hyperplane that separates the positive and the negative training instances in the feature space induced by the kernel function with the maximum margin and meanwhile, with the minimum misclassification error on the training instances.

The SVM training is equivalent to solving the following optimization problem:

$$\begin{aligned} \underset{\boldsymbol{w},\,\boldsymbol{\xi},\,b}{\operatorname{argmin}} \quad & \frac{1}{2}||\boldsymbol{w}||^2 + C\sum_{i=1}^{n}\xi_i \\ \text{subject to} \quad & y_i(\boldsymbol{w}\cdot\boldsymbol{x}_i + b) \geq 1 - \xi_i \\ & \xi_i \geq 0,\ \forall i \in \{1, ..., n\} \end{aligned} \tag{1}$$

where $\boldsymbol{w}$ is the normal vector of the hyperplane, C is the penalty parameter, $\boldsymbol{\xi}$ is the slack variables to tolerant some training examples falling in the wrong side of the hyperplane, and b is the bias of the hyperplane. The form of the optimization problem (1) is call the *primal form* of the SVM training.

To handle nonlinearly separable data, the above optimization problem is solved in dual form shown below where mapping functions can be easily applied.

$$\begin{aligned} \max_{\boldsymbol{\alpha}} . \quad & \sum_{i=1}^{n}\alpha_i - \frac{1}{2}\boldsymbol{\alpha}^T\boldsymbol{Q}\boldsymbol{\alpha} \\ \text{subject to} \quad & 0 \leq \alpha_i \leq C, \forall i \in \{1, ..., n\}, \\ & \sum_{i=1}^{n} y_i\alpha_i = 0 \end{aligned} \tag{2}$$

where α_i denotes the *weight* of $\boldsymbol{x}_i$; $\boldsymbol{Q}$ denotes an $n \times n$ matrix $[Q_{i,j}]$ and $Q_{i,j} = y_i y_j K(\boldsymbol{x}_i, \boldsymbol{x}_j)$, and $K(\boldsymbol{x}_i, \boldsymbol{x}_j)$ is a kernel value computed from a kernel function; C is for regularization. The kernel values of all the training instances form a *kernel matrix* [10]. The most common kernel functions are listed as follows.

- Gaussian: $K(\boldsymbol{x}_i, \boldsymbol{x}_j) = exp(-\gamma||\boldsymbol{x}_i - \boldsymbol{x}_j||^2)$
- Linear: $K(\boldsymbol{x}_i, \boldsymbol{x}_j) = \boldsymbol{x}_i \cdot \boldsymbol{x}_j$
- Polynomial: $K(\boldsymbol{x}_i, \boldsymbol{x}_j) = (a\boldsymbol{x}_i \cdot \boldsymbol{x}_j + r)^d$
- Sigmoid: $K(\boldsymbol{x}_i, \boldsymbol{x}_j) = \tanh(a\boldsymbol{x}_i \cdot \boldsymbol{x}_j + r)$

where γ, a, r and d are hyper-parameters of the kernel functions. The purpose of the kernel function is to implicitly

map the training data from their original data space to a higher dimensional data space. The mapping function $\varphi(\cdot)$ is implicit, because the mapping is embedded into the kernel function in the quadratic programming problem in the form of $K(\boldsymbol{x}_i, \boldsymbol{x}_j) = \varphi(\boldsymbol{x}_i) \cdot \varphi(\boldsymbol{x}_j)$. When training SVMs, we only need to specify the kernel function rather than specifying the mapping function $\varphi(\boldsymbol{x})$. When the training data is not linearly separable, practitioners often use kernel functions for mapping the training data to a higher dimensional data space.

2.1.1 The Sequential Minimal Optimization (SMO) algorithm

Platt [8] proposed the SMO algorithm for SVM training (i.e., solving the quadratic programming problem 2). SMO is simply a straightforward subspace-ascent algorithm restricted to two-dimensional subspaces [11], where subproblems are solved optimally. In SMO, a training instance $\boldsymbol{x}_i$ is associated with an optimality indicator f_i which is defined as follows.

$$f_i = \sum_{j=1}^{n} \alpha_j y_j K(\boldsymbol{x}_i, \boldsymbol{x}_j) - y_i \tag{3}$$

Training the binary SVM with SMO is to repeat the following three steps until the optimality condition is met.

Step 1: Find two training instances, denoted by $\boldsymbol{x}_u$ and $\boldsymbol{x}_l$, which have the maximum and minimum optimality indicators, respectively. The indexes of the two instances, denoted by u and l, can be computed as follows.

$$u = \underset{i}{\operatorname{argmin}}\{f_i | i \in I_u\} \tag{4}$$

and

$$l = \underset{i}{\operatorname{argmax}}\{\frac{(f_u - f_i)^2}{\eta_i} | f_u < f_i, i \in I_l\} \tag{5}$$

where $\eta_i = K(\boldsymbol{x}_u, \boldsymbol{x}_u) + K(\boldsymbol{x}_i, \boldsymbol{x}_i) - 2K(\boldsymbol{x}_u, \boldsymbol{x}_i)$; f_u and f_l are computed using Equation (3); I_u and I_l are defined as follows.

$$I_u = I_1 \cup I_2 \cup I_3,$$
$$I_l = I_1 \cup I_4 \cup I_5$$

and

$$I_1 = \{i | \boldsymbol{x}_i \in \mathcal{X}, 0 < \alpha_i < C\},$$
$$I_2 = \{i | \boldsymbol{x}_i \in \mathcal{X}, y_i = +1, \alpha_i = 0\},$$
$$I_3 = \{i | \boldsymbol{x}_i \in \mathcal{X}, y_i = -1, \alpha_i = C\},$$
$$I_4 = \{i | \boldsymbol{x}_i \in \mathcal{X}, y_i = +1, \alpha_i = C\},$$
$$I_5 = \{i | \boldsymbol{x}_i \in \mathcal{X}, y_i = -1, \alpha_i = 0\}.$$

A natural interpretation for these five sets is as follows. I_1 contains all the free support vectors; I_2 and I_5 include all the non-support vectors; I_3 and I_4 include all the on-bound support vectors.

Step 2: Improve the weights of $\boldsymbol{x}_u$ and $\boldsymbol{x}_l$, denoted by α_u and α_l, by updating them as follows.

$$\alpha_l' = \alpha_l + \frac{y_l(f_u - f_l)}{\eta} \tag{6}$$

and

$$\alpha_u' = \alpha_u + y_l y_u (\alpha_l - \alpha_l') \tag{7}$$

where $\eta = K(\boldsymbol{x}_u, \boldsymbol{x}_u) + K(\boldsymbol{x}_l, \boldsymbol{x}_l) - 2K(\boldsymbol{x}_u, \boldsymbol{x}_l)$. To guarantee the update is valid, when α_u' or α_l' exceeds the domain of $[0, C]$, α_u' and α_l' are adjusted into the domain.

Step 3: Update the optimality indicators of all instances. f_i is updated to f_i' using the following formula:

$$\begin{aligned} f_i' =& f_i + (\alpha_u' - \alpha_u) y_u K(\boldsymbol{x}_u, \boldsymbol{x}_i) \\ &+ (\alpha_l' - \alpha_l) y_l K(\boldsymbol{x}_l, \boldsymbol{x}_i) \end{aligned} \tag{8}$$

SMO repeats the above steps until the following condition is met.

$$f_u = \min\{f_i | i \in I_u\} \geq f_{max} \tag{9}$$

where f_{max} is computed as follows.

$$f_{max} = \max\{f_i | i \in I_l\} \tag{10}$$

The two selected instances in Step 1 together form a *working set*. SMO is simple and efficient, and therefore is widely used in LibSVM and many other SVM implementations [12], [13]. After the optimality condition is reached, the SVM can be used for prediction. The decision value of an instance $\boldsymbol{x}_i$, denoted by v_i, is computed as follows.

$$v_i = \sum_{j=1}^{n} y_j \alpha_j K(\boldsymbol{x}_j, \boldsymbol{x}_i) + b \tag{11}$$

where b is the bias of the hyperplane of the trained SVM. If v_i is greater or equal to 0, then the predicted class label is $+1$. Otherwise, the predicted class label is -1.

Algorithm 1 summarizes the whole training process using SMO. In Algorithm 1, $\boldsymbol{\mathcal{K}}_u$ and $\boldsymbol{\mathcal{K}}_l$ correspond to the u^{th} and the l^{th} rows of the kernel matrix, respectively.

Algorithm 1: SVM training use the SMO algorithm

Input: a training set $\mathcal{X}$ of n instances with labels $\boldsymbol{y}$
Output: a weight vector $\boldsymbol{\alpha}$

```
for i ← 1 to n do    /* initialize α and f */
    α_i ← 0, f_i ← −y_i
repeat
    search for f_u and u using Equation (4);
    compute kernel values K_u    /* u^th row */
    search for f_l and l using Equation (5);
    compute kernel values K_l    /* l^th row */
    update α_u and α_l using Equations (6) and (7);
    update f using Equation (8);
    search for f_max using Equation (10);
until f_u ≥ f_max
```

2.1.2 Computing probability

To convert the decision value v_i (cf. Equation 11) from a binary SVM to a probability, Platt [14] proposed to use a sigmoid function. The sigmoid is fitted to the SVM output, which ideally is a monotonic function of the probability. The intuition is that given $v_i > v_{i'} \geq 0$, the probability of $\boldsymbol{x}_i$ being a positive instance (i.e., with $+1$ label) should be greater than that of $\boldsymbol{x}_{i'}$ being a positive instance. This is because the instance $\boldsymbol{x}_i$ is further away from the hyperplane

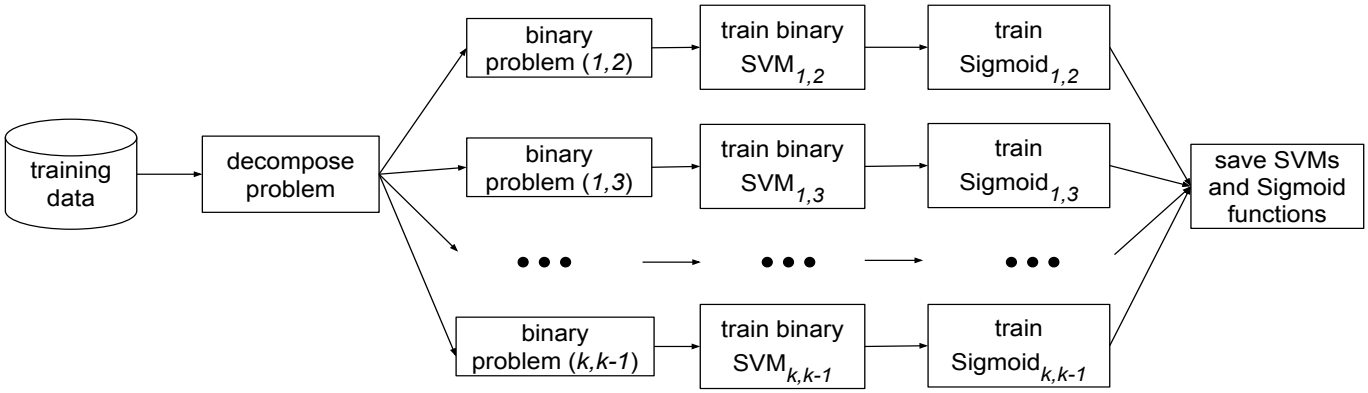


Fig. 1: Overview of MP-SVM Training

than $\boldsymbol{x}_{i'}$. Formally, the probability of an instance $\boldsymbol{x}_i$ being a positive instance is defined as follows.

$$P(y_i = 1|\boldsymbol{x}_i) = \frac{1}{1+\exp(Av_i + B)} \tag{12}$$

where v_i is computed by Equation (11), and the parameter A and B can be obtained by maximizing the following log likelihood.

$$\begin{aligned}\max_{A,B}. F = &\sum_{i=1}^{n} t_i \log(P(y_i = 1|\boldsymbol{x}_i)) \\ &- (1-t_i)\log(1 - P(y_i = 1|\boldsymbol{x}_i)) \\ \text{where } t_i = &\begin{cases} \frac{N_+ +1}{N_+ +2} & \text{if } y_i = +1 \\ \frac{1}{N_- +2} & \text{if } y_i = -1, \end{cases}\end{aligned} \tag{13}$$

N_+ denotes the number of positive training instances (i.e., $y_i = +1$), and N_- denotes the number of negative training instances (i.e., $y_i = -1$). Newton's method with backtracking is a commonly used approach to solve the above optimization problem [15] and is implemented in LibSVM.

2.2 Multi-class SVMs with probabilistic output (MP-SVMs)

In what follows, we present the training and prediction processes of MP-SVMs.

2.2.1 Training MP-SVMs

MP-SVMs are commonly implemented via pairwise coupling (also used in LibSVM) which has shown superiority over other methods [6]. During training MP-SVMs, many binary SVMs are trained using SMO and then their predicted values on the training instances are used to train the sigmoid function (cf. Equations 12 and 13) discussed above.

Figure 1 gives an overview of training MP-SVMs. Given the training dataset, the dataset is first decomposed into multiple subsets of two classes (i.e., binary problems). A binary problem (s,t) consists of all the instances of class s and t. There are $\frac{k(k-1)}{2}$ binary SVM classifiers in total, where k is the number of classes in the dataset. After training the binary SVM classifiers, the predicted values of $\text{SVM}_{s,t}$ on the binary problem (s,t) are used to train the corresponding sigmoid function to produce the binary probabilistic SVM. Finally, the trained SVMs and the sigmoid functions are saved, such that they can be used to predict the probability values of unseen data.

2.2.2 Prediction using MP-SVMs

When estimating probability, MP-SVMs first estimate local probability using binary SVMs with probability output and then estimate multi-class probability using all the local probabilities. More specifically, once we have the binary SVMs with probability output—also called a *local probability estimator*, the probabilities from all the local probability estimators for each instance are combined to obtain the multi-class probability. Suppose the dataset has k classes. Let us denote r_{st} to be the probability estimate of instance $\boldsymbol{x}_i$ (i.e., the local probability computed by Equation 12), where s and t denote the s^{th} class and t^{th} class, respectively. Multi-class probability estimation for an instance is formulated as follows.

$$\begin{aligned}&\min_{\boldsymbol{p}}. \sum_{s=1}^{k}\sum_{t:t\neq s}(r_{ts}p_s - r_{st}p_t)^2 \\ &\text{subject to} \sum_{i=s}^{k} p_s = 1 \text{ and } r_{ts} = (1 - r_{st})\end{aligned} \tag{14}$$

where p_s represents the probability of instance $\boldsymbol{x}_i$ belonging to the class s. The above problem is a convex quadratic problem with a linear equality constraint, and the problem can be solved by Gaussian elimination [16].

Figure 2 shows an overview of the prediction process of MP-SVMs. Given the testing data (which usually consist of unseen data during training), the $\frac{k(k-1)}{2}$ binary SVMs are used to predict the decision values of each instance. The predicted decision value by $\text{SVM}_{s,t}$ is plugged into the trained sigmoid function, $\text{Sigmoid}_{s,t}$ (cf. Equation 12), to obtain the probability of the instance belonging to class s. The probability values predicted by the $\frac{k(k-1)}{2}$ sigmoid functions are combined by solving optimization problem (14) to obtain the final multi-class probability.

Example 1. *Suppose we have three classes (i.e., $y_i \in \{1,2,3\}$) in total in a given problem, and three SVM classifiers with probabilistic output have been trained. The three SVM classifiers*

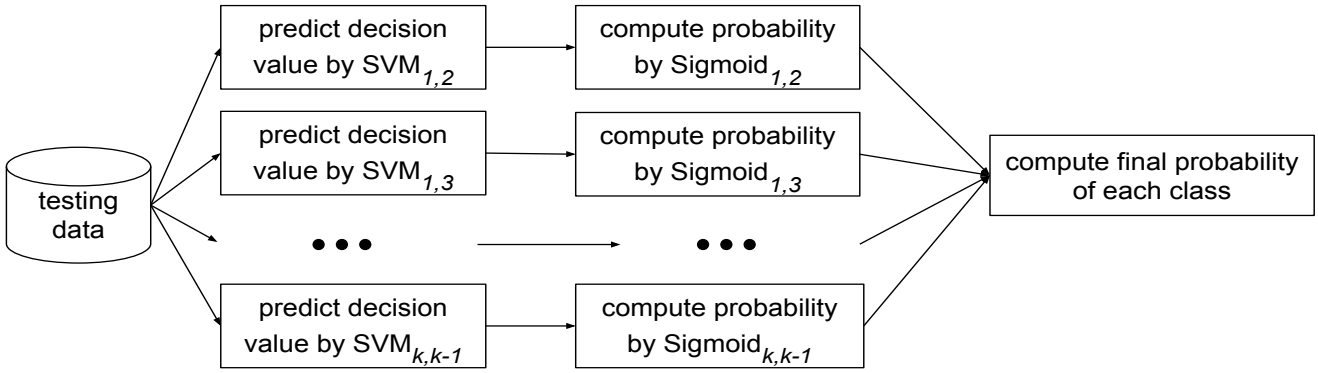


Fig. 2: Overview of MP-SVM Prediction

are denoted by $SVM_{1,2}$, $SVM_{1,3}$ and $SVM_{2,3}$, respectively. Given an unseen instance $\boldsymbol{x}_i$, the predicted probability of $\boldsymbol{x}_i$ belonging to class 1 is 0.8 by $SVM_{1,2}$; that belonging to class 3 is 0.4 by $SVM_{1,3}$; that belonging to class 2 is 0.4 by $SVM_{2,3}$. The three probability values are plugged into the optimization problem (14)*. After solving the optimization problem* (14)*, we obtain the probability of $\boldsymbol{x}_i$ belonging to class 1, 2 and 3 to be 0.85, 0.05 and 0.1, respectively.*

2.3 Graphics Processing Units

A GPU contains a large number of (e.g., thousands of) cores which are grouped into streaming multiprocessors (SMs). In the NVIDIA *Compute Unified Device Architecture* (CUDA), GPU threads are grouped into blocks which are also called *thread blocks*. Each thread block is executed in an SM. At any timestamp, an SM can only execute instructions of one thread block. Moreover, GPUs can run multiple programs (i.e., multiple CUDA kernel functions) concurrently, if the GPU has sufficient resources.

Compared with main memory, GPUs have relatively small memory (e.g., 12 GB memory in Tesla P100—a high-end GPU). The memory is called GPU global memory. Accessing the GPU global memory is much more expensive than computation, so we should avoid accessing the GPU global memory as much as possible. The data transfer between CPUs and GPUs is through PCI-e which is one order of magnitude slower than accessing the GPU global memory. Therefore, we should make full use of the GPU memory to efficiently handle large datasets, and reduce data transferring between CPUs and GPUs. In this paper, we address the limitation of GPU memory and take advantage of GPU massive computing capability.

3 Our solutions to MP-SVMs

In this section, we first discuss the challenges and the design rationale, and then propose a GPU baseline for multi-class SVMs with probabilistic output (*MP-SVMs* for short). Finally, we elaborate our highly efficient solution to MP-SVMs.

3.1 Challenges and design rationale

In the following, we discuss the challenges of accelerating the training and prediction algorithms for MP-SVMs, and present the design rationale of our algorithms.

3.1.1 Challenges

The key challenges of developing efficient MP-SVM algorithms are in two aspects: (i) efficiently access the kernel matrix and (ii) training many binary SVMs concurrently or estimating probability using many SVMs concurrently on GPUs.

First, the size of kernel matrix in the optimization problem of SVMs is quadratic in the number of training instances and is usually unable to be stored in memory. Therefore, the optimization problem cannot be solved at once. The SMO algorithm removes the constraint of having to store the whole kernel matrix in memory by decomposing the original problem to many subproblems. In each subproblem, only two rows of the kernel matrix need to be accessed. SMO has been proven being efficient on CPUs [17]. However, SMO needs to be redesigned to work efficiently on GPUs, because calculating two rows of kernel matrix for many times results in (i) lots of small read/write operations and (ii) many repeated kernel value computation, which dramatically slows down the SVM training process.

Second, training MP-SVMs requires training many binary SVMs, and estimating probability requires considering the results of many binary SVMs. There is a clear tradeoff on the parallel efficiency and memory consumption. Training the binary SVMs or estimating probability using binary SVMs sequentially underutilizes the GPU resources, while training all the binary SVMs concurrently or estimating probability using all the binary SVMs concurrently requires much larger memory footprint than the GPU memory.

3.1.2 Design rationale

In this paper, we aim to design an efficient GPU accelerated solution to MP-SVM by balancing memory consumption and the level of parallelism. To address the challenge of efficiently access the kernel matrix, we propose to select a larger working set, and reuse kernel values through a GPU memory buffer. We compute a number of rows of the kernel matrix in a batch, reuse the rows that are stored in the GPU memory buffer, and solve multiple subproblems in that batch. Thus, our algorithm avoids performing a large number of small read/write operations and reduce repeated kernel value computation. To address the challenge of concurrently training many binary SVMs or estimating probability using many SVMs on GPUs, we propose kernel

value and support vector sharing technique among the binary SVMs to reduce the memory usage.

In the following, we start with the naive solution (GPU baseline), and then present the design of our solution called "GMP-SVM" in more details.

3.2 The GPU baseline

Recall that MP-SVMs consist of multiple binary SVMs. A naive approach is to train the binary SVMs on the GPU one by one, and to estimate probability for multiple instances using one binary SVM at a time. The GPU baseline consists of three phrases: **Phase (i)** training binary SVM classifiers one by one using SMO, **Phase (ii)** adapting each SVM classifier to a local probability estimator (cf. Equations 12 and 13), and **Phase (iii)** estimating probabilities by combining the results from all the local probability estimators (cf. solving problem 14).

Phase (i): When training a binary SVM, Step 1 and Step 3 of SMO can be done in parallel, while Step 2 cannot be parallelized because the two weights are dependent (cf. Section 2.1). Step 1 is essentially searching for the minimum or maximum value from an array, and can be done in parallel on GPUs using parallel reduction [18], where each thread compares two elements and discards the larger/smaller one until only one element is left in the array. Step 3 can also be done in parallel, where each thread updates an optimality indicator using Equation (8).

Phase (ii): Converting a binary SVM classifier to a local probability estimator is to learn the parameters A and B for a sigmoid function based on the training data. Learning the parameters A and B (cf. Equation (12)) is to solve Problem (13) using Newton's method with backtracking. The major computation of the Newton's method with backtracking is computing the value and the gradients of the objective function of Problem (13). Computing the value of the objective function is effectively computing the sum of n elements, and is done by parallel reduction on GPUs. The gradients with respected to A and B are computed by:

$$\frac{\partial F}{\partial A} = \sum_{i=1}^{n} v_i(t_i - P(y_i = 1|\boldsymbol{x}_i))$$
$$\frac{\partial F}{\partial B} = \sum_{i=1}^{n} (t_i - P(y_i = 1|\boldsymbol{x}_i))$$

The above gradients are also effectively the sum of n elements, and can also be done by parallel reduction.

Phase (iii): After we have obtained all the local probability estimator, we can perform the multi-class probability estimation using those local probability estimators. Here, we present techniques to (1) compute the decision values in parallel, (2) compute multi-class probability of one instance in parallel, and (3) compute multi-class probability of multiple instances in parallel.

(1) Compute decision values in parallel: As discussed in Section 2.2, we need to use the decision values computed by each binary SVM to estimate local probability (i.e., Equation (12)). Given n training instances and $\frac{k(k-1)}{2}$ binary SVMs, we need to compute $n \times \frac{k(k-1)}{2}$ decision values, where k is the number of classes.

The training instances are independent and the binary SVMs are also independent. Theoretically, all the binary SVMs can be used to predict the decision values simultaneously. However, due to the GPU memory limitation, we compute decision values of multiple training instances concurrently using one binary SVM at a time. We will present a better algorithm in the later section with support vector and kernel value sharing. Given an instance $\boldsymbol{x}_i$, we allocate a thread block to compute the sum of Equation (11) and to obtain the decision value v_i. In total, we dedicate $n \times \frac{k(k-1)}{2}$ thread blocks for computing all the decision values. If $n \times \frac{k(k-1)}{2}$ is larger than the maximum number of blocks that the GPU can support, we divide the blocks into a few groups and launch one group of blocks at a time.

(2) Compute multi-class probability of one instance in parallel: We use the method derived by Wu et al. [16] to solve Problem (14). The multi-class probability $\boldsymbol{p} = \langle p_1, p_2, ..., p_k \rangle$ is computed as follows.

$$\boldsymbol{p} = \frac{\boldsymbol{Q}^{-1}\boldsymbol{e}}{\boldsymbol{e}^T\boldsymbol{Q}^{-1}\boldsymbol{e}}, \quad \text{where } Q_{st} = \begin{cases} \sum_{u:u\neq s} r_{us}^2 & \text{if } s = t \\ -r_{st}r_{ts} & \text{sf } s \neq t \end{cases} \tag{15}$$

where $\boldsymbol{e}$ is a $k \times 1$ vector of all ones; s and t denote the s^{th} class and t^{th} class, respectively; $\boldsymbol{Q}$ is positive semi-definite. A small value is added to $\boldsymbol{Q}$ when its inversion does not exist. We propose to use the matrix operation library (i.e., cuSparse [19]) to exploit GPUs, as Equation (15) mainly involves sparse matrix operations.

(3) Compute multi-class probability of multiple instances in parallel: The above process is for estimating probability $\boldsymbol{p} = \langle p_1, p_2, ..., p_k \rangle$ for one instance. We launch multiple procedures on GPUs for estimating probability for multiple instances, to take advantage of GPUs' high performance.

Problems of the GPU baseline: The GPU baseline is notably more efficient than LibSVM with OpenMP. However, two key unsolved problems are: (i) many kernel values are repeatedly computed because an SVM is trained iteratively, which results in repeated accesses to the high latency GPU memory; (ii) training one binary SVM (or estimating probability using one binary SVM) at a time underutilizes the GPU hardware.

3.3 Our GMP-SVM solution

To reduce repeated memory accesses, kernel value reusing and sharing can be an effective approach. To improve the GPU utilization, it is natural to consider training multiple binary SVMs (and estimating probabilities using multiple binary SVMs) concurrently. However, reusing kernel values is tricky, because greedily reusing kernel values may result in local optimization on the working set. Sharing kernel values among binary SVMs requires organizing the kernel values properly. Training many binary SVMs (or estimating probability using multiple binary SVMs) at a time requires a much larger memory footprint than the GPU memory.

To address the challenges, we develop a novel solution called ***GMP-SVM*** ("G" stands for "GPU") with two-level optimization for training MP-SVMs and high parallelism

for estimating probability. GMP-SVM reduces high latency memory accesses and memory consumption through batch processing, kernel value reusing and sharing, and support vector sharing. In the binary SVM level, we use a larger working set of violating instances than the SMO solver does (i.e., SMO selects two instances), such that we can amortize the data access overhead on GPU memory and solve SMO subproblems in a batch. We (i) precompute all the kernel values for the violating instances in a batch to achieve cheaper cost per kernel value, (ii) store the kernel values to a GPU buffer which stores all the kernel values in the working set, and (iii) optimize the SVM on the working set with an approach to reduce the negative effect of local optimization on the working set. In the MP-SVM level, we concurrently train multiple binary SVMs with kernel value sharing among the binary SVMs. When estimating probabilities, GMP-SVM concurrently computes the probabilities for multiple instances using multiple binary SVMs with support vector and kernel value sharing. With the two level optimization for training and support vector and kernel value sharing for estimating probability, we address the memory access and GPU utilization problems of the GPU baseline.

3.3.1 Techniques used in binary SVM level

As we have mentioned in Section 2.1, the SMO algorithm selects two training instances (which together form a working set) to improve the current SVM. For the working set, SMO needs to compute all the related kernel values, i.e., two rows of the kernel matrix (an example of kernel matrix is available in Section 2.1). The kernel values are later used for adjusting the currently trained SVM. It is known that the kernel value computation is the bottleneck of the SVM training algorithm [18], [20] when the data set is large. Although GPUs can compute a row of the kernel matrix faster than CPUs, it still takes majority of the training time according to our experiments in the SMO algorithm. This is true for high dimensional datasets which are emerging in many applications. Therefore, we propose techniques to efficiently compute the kernel values in batches, and to reuse and share kernel values.

Instead of using a working set of size two, we propose to use a bigger working set and solve multiple subproblems of SMO in a batch. We precompute all the kernel values for the working set and store them in a GPU buffer which is a preallocated space on the GPU global memory. In each time we update the working set, q (where $q \geq 2$) instances in the working set will be replaced with q new violating instances, such that (i) q rows of the kernel matrix can be computed in one execution to make efficient use of the GPU and reduce GPU memory accesses, and (ii) the kernel values in the GPU buffer can be reused (i.e., GPU buffer size is larger than q). Our experimental results show that when $q > 10$, the computation cost per row is often over ten times cheaper than the cost of computing a row individually, due to the massive parallelism of GPUs.

It is worthy to point out that solving $q/2$ subproblems in a batch is cheaper than solving the same number of subproblems individually in the original SMO algorithm. The cheaper computation cost is due to the batch processing and reuse of the kernel values. One possibility is that solving the $q/2$ subproblems in a batch may lead to more training iterations compared to solving the subproblems in the traditional way. However, the efficiency of the whole training is improved in practice, because the cost of the extra training iterations is much lower than the cost saved in kernel value computation. Our empirical results in Section 4 will confirm this.

In the following, we first discuss our approach to select the q violating instances to refresh the working set. Then, we provide more details of computing kernel values for the q instances and store them to a GPU buffer. Last, we present a technique to reduce the negative effect of local optimization on the working set.

Selecting q violating instances: Our intuition for updating the working set is to choose q training instances that violate the optimality condition (cf. Constrain (9)) the most, such that the current SVM can be potentially improved the most [21]. The violation to the optimality condition is measured by the optimality indicators (i.e., cf. Equation (3)). Hence, we first sort the training instances based on their optimality indicators in ascending order. Then, we choose the top $\frac{q}{2}$ training instances whose $y_i\alpha_i$ can be increased; and we choose the bottom $\frac{q}{2}$ training instances whose $y_i\alpha_i$ can be decreased. We consider both y_i and α_i for each training instance, because of the constraints $\sum y_i\alpha_i = 0$ and $0 \leq \alpha_i \leq C$ in Problem (2). By choosing the working set in this way, the search space for the α values of the q instances is large, and as a result the objective value of Problem (2) is likely to increase the most. After choosing the working set, we solve the optimization problem formed by this working set using SMO.

One implementation detail we would like to note here is that keeping half of the violating instances in the previous batch leads to faster convergence (i.e., adding only $\frac{q}{2}$ violating instances to form a new working set). The reason may be due to the avoidance of the local optimization on the working set as we will discuss later.

Maintaining a GPU buffer for kernel values: Before we solve the optimization problem formed by the q violating instances, we pre-compute all the kernel values related to the q instances (i.e., q rows of the kernel matrix). This is because the q rows of the kernel matrix are repeatedly used during the optimization. It is important to point out that the q rows of the kernel values only consume a small amount of memory, where q is user defined variable and is usually smaller than 1024 in practice. Computing those kernel values is essentially matrix multiplication between the q instances and the rest of the training instances, because computing a kernel value can be viewed as a dot product of two vectors [12]. Thus, the kernel value computation here can be efficiently carried out by the cuSPARSE library [19].

The kernel values computed here are stored in a GPU buffer on the GPU global memory, as the kernel values will be repeatedly used by SMO while improving the current SVM with the updated working set. Note that the SMO in our algorithm only considers the instances in our working set, which is different from the original SMO that needs to consider all the training instances in every iteration. As a result, one iteration of the SMO in our algorithm is often

much cheaper than the traditional SMO.

To store the pre-computed kernel values, we allocate a GPU memory buffer which can store $m \times q$ rows of the kernel matrix (i.e., allow m batches to be stored in the buffer). The first-in first-out batch replacement strategy is used when the buffer is full. Although other strategies may be more effective, we find first-in first-out simple and sufficiently effective in our algorithm. Finding the best strategy for replacement is out of the scope of this paper.

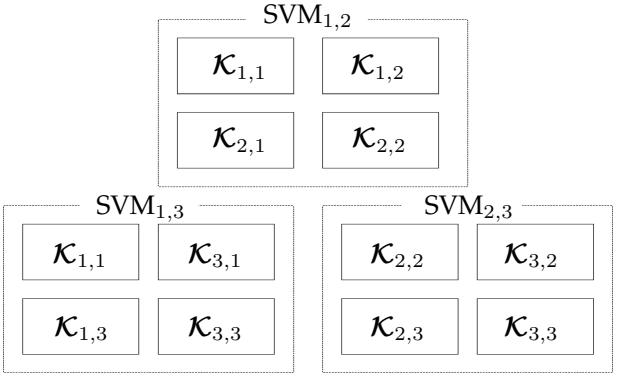
Reducing the negative effect of local optimization on the working set: We improve the currently trained SVM classifier based on the updated working set using SMO. One approach is to improve the current SVM until no further progress can be made using the current working set. However, such an approach results in local optimization on the working set, i.e., the SVM tends to classify the instances in the working set accurately but misclassify other instances. To mitigate the local optimization problem, we terminate the improvement process earlier and shift to the next working set for further improvement. Specifically, we propose to use $\delta = (f_l - f_u)$ to decide when to terminate. Note that δ indicates how far away the current SVM to the optimal (cf. Constraint (9)). If δ is large, then we improve the current SVM by fewer iterations; otherwise, we improve the SVM by more iterations.

The above techniques significantly improve the efficiency on the binary SVM training, as we will see in the empirical results in Section 4.

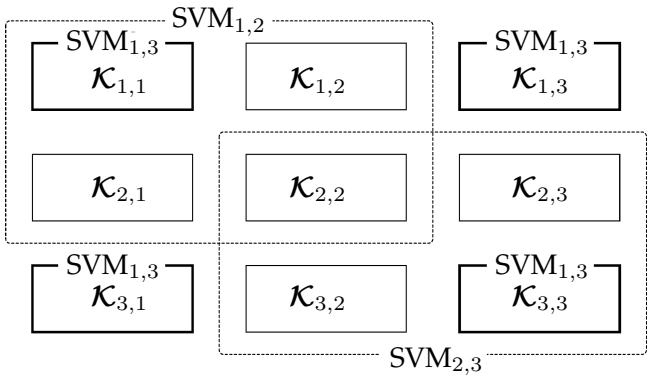
3.3.2 Techniques used in the MP-SVM level

As discussed earlier, we need to train $\frac{k(k-1)}{2}$ binary SVM classifiers, where k is the number of classes in the training dataset. An observation is that any two binary SVM classifiers are independent. Therefore, we can simultaneously train several binary SVMs. However, due to the limited GPU memory and streaming multiprocessors, we cannot train all the binary SVMs at once. When the training dataset is large, the whole GPU may be occupied by only one binary SVM training. To allow more binary SVMs to be trained concurrently, we limit the number of streaming multiprocessors (SMs) that each binary SVM can use. Specifically, we use larger GPU thread blocks, such that the total number of blocks for a binary SVM is smaller than the number of SMs.

Moreover, we observe that the kernel values among binary SVMs can be shared. To explain this, let us see an example shown in Figure 3a where the number of classes is three (i.e., $k = 3$) and the class labels are 1, 2 and 3, respectively. As we can see from the figure, the three binary SVMs each has an independent kernel matrix, where $\boldsymbol{\mathcal{K}}_{s,t}$ represents kernel values computed from $K(\boldsymbol{x}_i, \boldsymbol{x}_j)$ for all $\boldsymbol{x}_i$ and $\boldsymbol{x}_j$ in classes s and t, respectively. Figure 3b shows the potential of sharing kernel values among the three binary SVMs. The upper left dashed box contains the kernel matrix for $\text{SVM}_{1,2}$, and the bottom right dashed box contains the kernel matrix for $\text{SVM}_{2,3}$. The kernel matrix for $\text{SVM}_{1,3}$ is the four blocks at the corners of the figure; the four blocks together form the kernel matrix for $\text{SVM}_{1,3}$. As we can see from the two figures, we reduce the number of blocks from 12 to 9 through sharing. The kernel value sharing technique is useful in two aspects. First, the technique helps reduce the GPU memory consumption, and hence allows more binary SVMs to be trained concurrently. Second, the technique allows our algorithm to compute fewer kernel values, and hence reduces GPU memory accesses.



(a) Kernel matrices of 3 binary SVMs



(b) Kernel matrices shared by 3 binary SVMs

Fig. 3: Organizing the kernel matrices of binary SVMs

Adapting binary SVMs to local probability estimators: We improve the GPU baseline in two aspects: (i) we learn multiple sigmoid functions concurrently; (ii) we evaluate multiple possible values for A and B concurrently in the Newton's method, instead of evaluating only one possible value as the GPU baseline. We have enough GPU resources for these highly parallel operations, thanks to our kernel value sharing and reusing.

Algorithm 2 summarises the steps of the training algorithm of GMP-SVM. For clarity, we write two for-loops in the pseudo-code, which in fact are parallelized. In our implementation, we concurrently train multiple binary SVMs on GPUs (i.e. Lines 3 to 11) and kernel values among the SVMs are shared during training. The SMO solver with a second order heuristic [17] is used to improve the SVM in Line 9. Line 12 denotes a parallel procedure for computing decision values and will be further discussed next, as the procedure is also used in probability estimation.

3.3.3 Estimating probability using kernel value sharing

Concurrently estimating probabilities for multiple instances using multiple binary SVMs requires more memory than the GPU memory footprint. To enable decision value prediction using multiple binary SVMs, we propose to share support vectors between SVMs and to share kernel values while computing the decision values. The reason behind support

Algorithm 2: Training MP-SVMs

Input: Training set $\mathcal{X}$ of k classes
Output: $\boldsymbol{S}$ of $k(k-1)/2$ binary SVM classifiers

```
/* all instances with label s                    */
for s = 1 to k do
    /* all ins. with label t                     */
    for t = s + 1 to k do
        y, X_{s,t} ← getIns(X, s, t)   /* y are labels */
        α ← 0, f ← −y, ws ← φ
        repeat
            sort(f)   /* sort f ascendingly      */
            vioIns ← selectViolatingInstance(q, f)
            vioKVal ← compKernelVal(vioIns, X_{s,t})
            ws ← updateWorkingSet(vioIns, vioKVal)
            α ← ImproveSVM(α, y, f, ws)
            /* use Equation (5)                  */
            f ← updateF(α, y, f, X_{s,t})
        until f_u ≥ f_l
        /* use Equation (7)                      */
        v ← compDecisionVal(α, X_{s,t})
        /* solve Problem (9)                     */
        A, B ← trainSigmoid(v)
        S ← S ∪ saveSVM(α, A, B)
```

vector sharing is that the training datasets of two binary SVMs may have more than a half of the training instances in common, and some common training instances may become support vectors which can be shared between the SVMs. In fact, without support vector sharing, the same training instance may be stored in $(k-1)$ binary SVMs as a support vector. Our support vector sharing technique reduces the GPU memory consumption by up to a factor of $(k-1)$. Moreover, from Equation (11), we need to compute kernel values for the instance $\boldsymbol{x}_i$ with all the support vectors (i.e., training instance with $\alpha > 0$) of all the binary SVMs. The kernel values involved in Equation (11) can be shared among the SVMs if the SVMs share support vectors.

4 Experimental Study

In this section, we empirically evaluate the performance of GMP-SVM. We conducted all of our experiments on a workstation running Linux with two Xeon E5-2640 v4 10 core CPUs, 256GB main memory and an NVIDIA Tesla P100 GPU of 12GB memory. Our GMP-SVM solution and the GPU baseline are implemented in CUDA-C. We also implemented our GMP-SVM solution in C++ to obtain its CPU version denoted by **CMP-SVM** in order to study the impact of parallel algorithm design on the CPU. Both CMP-SVM and LibSVM use the two Xeon E5-2640 v4 10 core CPUs, and GMP-SVM mainly uses the NVIDIA Tesla P100 GPU. The code is available at https://github.com/zeyiwen/thundersvm. We used Gaussian kernel and hyper-parameters C and γ for the kernel on each dataset are the same as the existing studies [18], [20], [22]. We compare GMP-SVM with our GPU baseline, our CPU version of

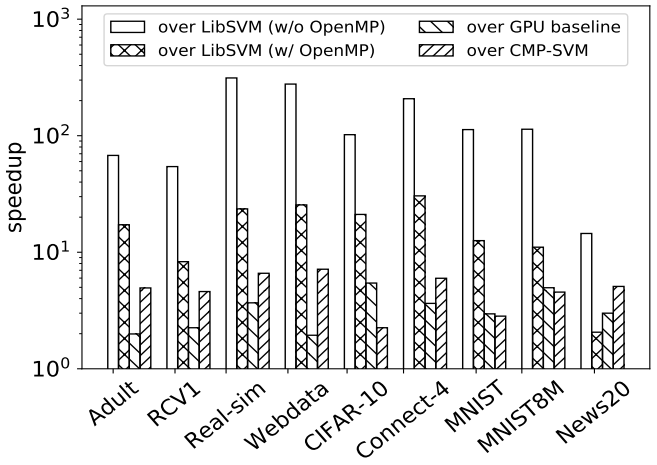


Fig. 4: Training time speedup of GMP-SVM over other MP-SVM implementations

GMP-SVM, LibSVM (with and without OpenMP). Table 2 gives the details of the datasets which are publicly available (e.g., LibSVM website). The first four datasets have two classes, and are used for studying the effectiveness of our techniques used in binary SVM level. The remaining five datasets are used for studying the whole GMP-SVM solution. For a sanity check, we also compare GMP-SVM with GTSVM [20], OHD-SVM [23] and GPUSVM [12] which are arguably the state-of-the-art GPU library for multi-class SVMs and binary SVMs, respectively, although OHD-SVM only supports binary SVMs and GTSVM does not support MP-SVMs and cannot be extended to train MP-SVMs (more discussion in Section 5).

4.1 Efficiency and MP-SVM classifier comparison

We study the performance of our GMP-SVM, the GPU baseline, our CMP-SVM and LibSVM. We set the GPU buffer size to 1024 (i.e., store 1024 rows of the kernel matrix), and q to 512 for GMP-SVM. LibSVM with OpenMP uses 40 threads, which achieves the best performance. The GPU baseline uses 4GB of GPU memory for kernel value caching. Our CMP-SVM also uses 40 CPU threads, which achieves the best efficiency.

Efficiency comparison: Figure 4 shows the speedup on training of GMP-SVM over other implementations including the GPU baseline, CMP-SVM and LibSVM. As we can see from the results, GMP-SVM consistently outperforms LibSVM without OpenMP by one to two orders of magnitude, LibSVM with OpenMP by 10x times, and the GPU baseline by two to five times. For example, when training MP-SVMs for MNIST8M, GMP-SVM only takes two hours, while the GPU baseline takes about ten hours. Another observation on the datasets with 2 classes is that GMP-SVM is the same as the GPU baseline when estimating probability. This is because only one SVM classifier exists and GMP-SVM cannot benefit from concurrently running multiple binary SVMs for prediction. Finally, when comparing our algorithm implemented for GPUs and CPUs, we can observe that GMP-SVM consistent outperforms CMP-SVM by three to ten times. This demonstrates the importance of using GPUs for our MP-SVM solution. Better GPUs such as

TABLE 2: Dataset information and parameters

Dataset	Adult	RCV1	Real-sim	Webdata	CIFAR-10	Connect-4	MNIST	MNIST8M	News20
# of classes	2	2	2	2	10	3	10	10	20
cardinality	32,561	20,242	72,309	49,749	50,000	67,557	60,000	8,100,000	15,935
dimension	123	47,236	20,958	300	3,072	126	780	784	62,061
C	100	100	4	10	10	1	10	1000	4
γ	0.5	0.125	0.5	0.5	0.002	0.3	0.125	0.006	0.5

TABLE 3: Efficiency comparison among LibSVM, GPU baseline, and our GMP-SVM

Dataset	elapsed time (sec)									
	LibSVM w/o OpenMP		LibSVM w/ OpenMP		GPU baseline		our CMP-SVM		our GMP-SVM	
	train	predict	train	predict	train	predict	train	predict	train	predict
Adult	164.62	75.4	41.81	6.29	4.83	0.29	12.01	1.53	2.43	0.29
RCV1	54.87	80.58	8.38	4.31	2.27	0.16	4.65	1.97	1.01	0.16
Real-sim	986.1	488.49	74.21	25.57	11.58	0.75	20.8	5.83	3.15	0.75
Webdata	734.92	188.27	67.51	12.88	5.13	0.57	18.95	2.77	2.65	0.57
CIFAR-10	22,523	19,090	4,659	3,074	1,200.2	212.5	495.65	89.79	220.7	29.3
Connect-4	1,212.69	539.2	177.9	39.2	21.24	2.17	34.92	6.29	5.84	0.86
MNIST	3,847.96	3,113.16	429.1	245.7	100.83	30.46	96.55	12.04	34.10	4.62
MNIST8M	810,794	1.2x10^6	78,856	79,840	35,390	2,945.87	32,470.37	1675.24	7,134.12	927.06
News20	237.68	79.37	33.77	14.25	49.15	16.93	83.49	2.83	16.4	0.52

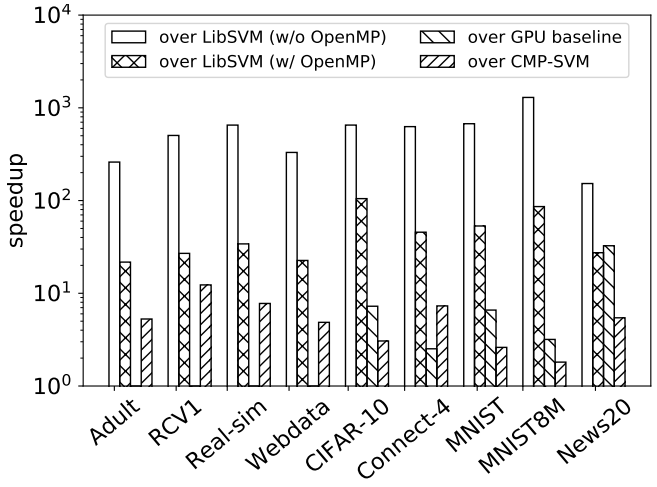


Fig. 5: Prediction speedup of GMP-SVM over other MP-SVM implementations

V100 should further improve the efficiency of GMP-SVM, due to higher memory bandwidth and more cores.

Figure 5 shows the results of speedup on prediction of GMP-SVM over other probabilistic SVM implementations. As we can see from the figure, GMP-SVM consistently outperforms LibSVM without OpenMP by two orders of magnitude. When OpenMP is used for LibSVM, GMP-SVM still outperforms it by more than 10 times. GMP-SVM performs similarly with the GPU baseline for the four datasets (i.e., Adult, RCV1, Real-sim and Webdata) with two classes, because GMP-SVM is in fact the same as the GPU baseline when handling binary problems. Therefore, GMP-SVM has no speedup over the GPU baseline for the first four datasets shown in Figure 5. When handling multi-class problems (e.g., datasets MNIST and News20), GMP-SVM is 3 to 30 times faster than the GPU baseline, thanks to the techniques (e.g., sharing kernel values and training multiple binary SVMs concurrently) proposed in Section 3.3. In comparison with CMP-SVM, GMP-SVM achieves 2 to 8 times speedup thanks to the high parallelism of GPUs.

The detailed elapsed time on both training and prediction of each implementation of each dataset is shown in Table 3.

Classifier comparison: We should expect to see that GMP-SVM and LibSVM produce identical SVMs, because GMP-SVM can be viewed as a highly parallelized version of LibSVM. We have measured the training error to confirm if GMP-SVM produces identical results as LibSVM, and the results are shown in Table 4. As we can see from the results, the training and prediction errors are identical, which implies that GMP-SVM and LibSVM produce the same SVMs. The prediction error is computed using the test set for the corresponding training problem. When the test set is not available, we use the training dataset to serve as the test set. To further confirm the SVMs trained by GMP-SVM and LibSVM are the same, we also compare the bias of the trained MP-SVMs, and the results are shown in Column "bias" of Table 4. Note that we used the bias of the last binary SVM for the multi-class problems. As we can see from the results, the biases of SVMs trained by GMP-SVM are the same to those of LibSVM. Note that existing studies in machine learning commonly compare the difference of $||\boldsymbol{w}||$ of two algorithms. However, it is impossible for kernelized SVMs, because $||\boldsymbol{w}||$ is in an unknown data space. Finally, we also varied the hyper-parameters C from 0.01 to 100 and γ from 0.03 to 10 on all the datasets, and compared the training/prediction errors and bias between LibSVM and GMP-SVM. The results again confirm that GMP-SVM and LibSVM produce identical classifiers.

4.2 Sensitivity studies

As we have discussed in Section 3.3.1, kernel value computation takes most of the training time. Here, we investigate the following two techniques closely related to kernel value computation. We use four datasets as representatives in this set of experiments: two for binary SVM training and two for multi-class SVM training.

Effect of the GPU buffer for kernel values: We study the effect of the GPU buffer size on the overall training time. Note that changing the GPU buffer size is effectively varying

TABLE 4: Final classifier comparison between LibSVM and GMP-SVM

Dataset	bias term of the decision function		training error		prediction error	
	LibSVM	GMP-SVM	LibSVM	GMP-SVM	LibSVM	GMP-SVM
Adult	-0.510	-0.510	4.4%	4.4%	17.3%	17.3%
RCV1	-0.512	-0.512	0.11%	0.11%	4.2%	4.2%
Real-sim	-1.061	-1.061	0.27%	0.27%	0.27%	0.27%
Webdata	-0.936	-0.947	0.54%	0.54%	0.56%	0.56%
CIFAR-10	0.0245	0.0245	0.35%	0.35%	0.35%	0.35%
Connect-4	0.233	0.233	4.39%	4.39%	4.39%	4.39%
MNIST	0.360	0.360	0%	0%	10.18%	10.18%
MNIST8M	-7.339	-7.339	0%	0%	0%	0%
News20	-0.0016	-0.0016	2.25%	2.25%	15.7%	15.7%

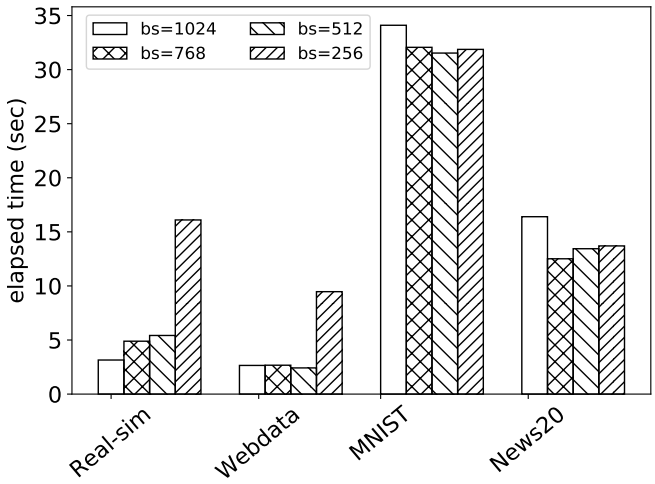


Fig. 6: Varying buffer size

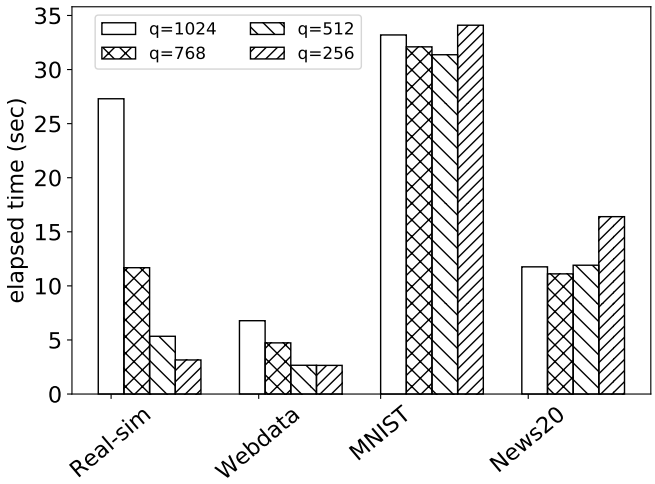


Fig. 7: Varying # of violating instances

the size of the working set. As we can see from Figure 6, the medium size GPU buffer (e.g., bs=512) achieves competitive outcomes. Generally speaking, larger buffer size leads to better performance. The reason is that larger buffer allows more kernel values to be reused. However, when the buffer size is too big, the working set tends to contain many not useful training instances, which puts more burden when improving the current SVMs.

Effect of the number of new violating instances: We conducted experiments to study the effect of varying q (i.e., the number of violating instances). According to the result in Figure 7, q should be about 1/2 of the GPU buffer size. This is because large q results in flushing out all the kernel values in the GPU buffer, while small q leads to more expensive cost per kernel value (i.e., the batch size for the kernel value computation is too small).

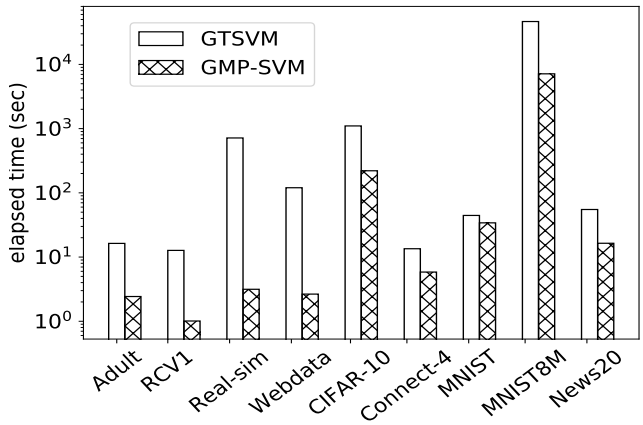


Fig. 8: Training time of GMP-SVM and GTSVM

4.3 Comparison with a third party GPU SVM library

For a sanity check, we compare GMP-SVM with two most recent GPU implementations for SVMs: GTSVM and OHD-SVM. The two implementations are arguably the state-of-the-art GPU based SVMs. GTSVM is for multi-class SVM training and OHD-SVM is for binary SVM training.

4.3.1 Comparison with GTSVM

We first compare GMP-SVM with GTSVM [20] which can train binary and multi-class SVMs but does not support MP-SVMs (more discussion in Section 5). Although our GMP-SVM is mainly for solving MP-SVM problems, GMP-SVM can be used to train binary and multi-class SVMs. We compare GTSVM with GMP-SVM regarding the total elapsed time for training multi-class SVMs. The results are shown in Figure 8. According to the result, GMP-SVM consistently outperforms GTSVM often by about five times on all the nine datasets, thanks to our optimizations when training (multiple) binary SVMs and kernel value sharing techniques.

4.3.2 Comparison with OHD-SVM and GPUSVM

OHD-SVM has been proposed recently [23], but it is for binary SVM training only. Therefore, we compare GMP-SVM with OHD-SVM using the four datasets which contain only two classes. The results are shown in Figure 9.

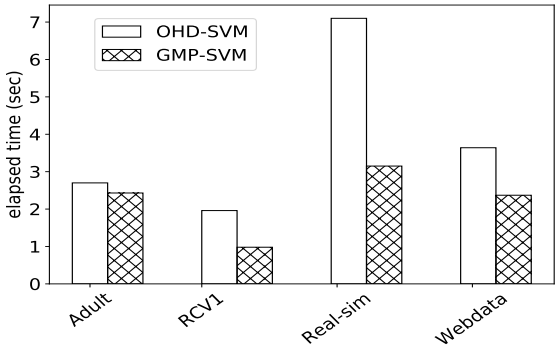


Fig. 9: Training time of GMP-SVM and OHD-SVM

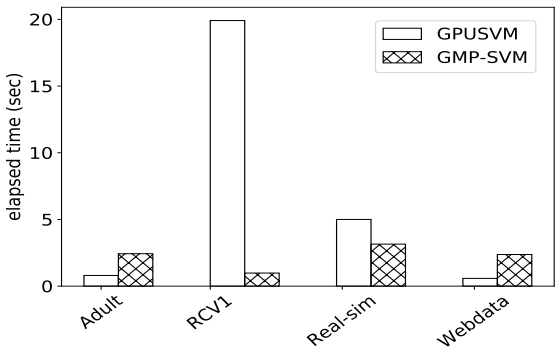


Fig. 10: Training time of GMP-SVM and GPUSVM

In addition to supporting multi-class probabilistic SVMs, GMP-SVM consistently outperforms OHD-SVM, thanks to our optimization on the binary SVM training level.

We also compare GMP-SVM with GPUSVM [12] in Figure 10. Please note that GPUSVM does not support multi-class classification and probabilistic output. Hence, we only use the four datasets which contain only two classes. As we can see from the figure, GMP-SVM significantly outperforms GPUSVM in large datasets, and achieves similar efficiency in small datasets. We have noticed that GPUSVM uses the dense data representation, which leads to higher computation cost for large datasets and also requires more memory to store the training data. This is the key reason why GPUSVM is much slower than GMP-SVM on the *RCV1* dataset.

4.4 Elapsed time on different components of GMP-SVM

For a better understanding of GMP-SVM and identifying potential improvement of GMP-SVM in the future, we measure the elapsed time of different components of GMP-SVM during training and prediction. Please note that the total elapsed time for GMP-SVM training and prediction is available in Table 3. Therefore, we only present the percentage of the total elapsed time of each component.

4.4.1 Time taken on each component in GMP-SVM training

The key components of GMP-SVM training include (i) kernel value computation, (ii) solving the subproblem (i.e., solve the optimization problem that consists of instances in the working set), and (iii) the remaining tasks such as selecting the working set and updating the optimality indicators.

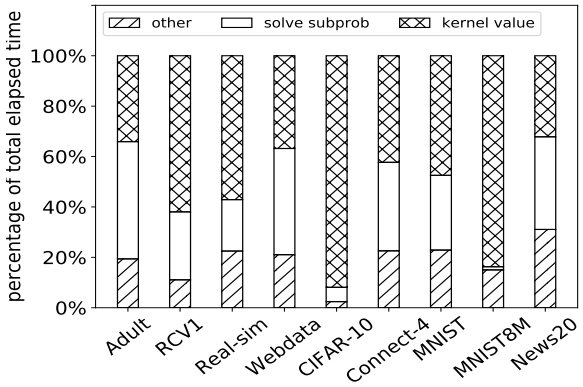


Fig. 11: Percentage of elapsed time of GMP-SVM training

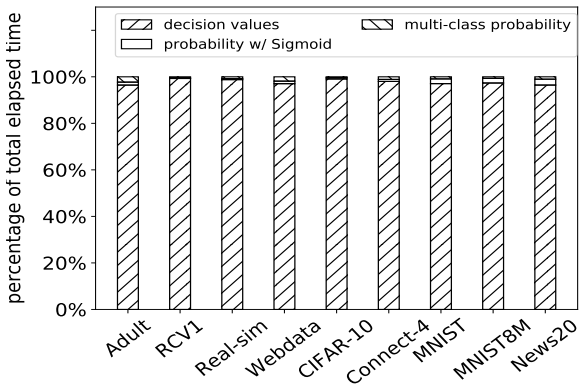


Fig. 12: Percentage of elapsed time of GMP-SVM prediction

Figure 11 shows the result. As we can see from the result, kernel value computation tends to dominate the whole training process, and solving the subproblem is the second most expensive process. The other tasks consume roughly 20% of the total training time. This result gives the insight of GMP-SVM, and the further improvement of GMP-SVM training should focus on improving kernel value computation and solving the subproblem.

4.4.2 Time on each component in GMP-SVM prediction

The key components of GMP-SVM prediction include (i) computing the decision values (cf. Equation 11), (ii) computing the probability values using sigmoid functions (cf. Equation 12), and (iii) computing the multi-class probability values (cf. Equation 14).

Figure 12 depicts the results. As we can see from the figure, computing the decision values dominates the whole prediction process. In comparison, the cost of solving the optimization problem (14) using Equation 15 for obtaining the multi-class probability is negligible. This insight is helpful to us and other researchers who want to further improve GMP-SVM prediction in the future.

5 Related work

Machine learning has been successful in many applications in recent years [24], [25], [26], and high-performance computing has played a key role in this success. This study mainly focuses on improving the efficiency of a machine

learning algorithm: multi-class probabilistic SVMs. Many existing studies have been dedicated to training SVMs. In what follows, we categorize the most relevant related work into two categories: the studies dedicated to training SVMs using CPUs and the studies dedicated to training SVMs using GPUs.

Training SVMs using CPUs: Platt [8] proposed the SMO algorithm which is simple and efficient, and hence SMO is used in LibSVM, WEKA [27] and Catanzaro's algorithm [12]. Other studies in training linear SVMs, such as Joachims' algorithm using cutting plane [28] and "Pegasos" [29] cannot handle non-linear kernels and does not support MP-SVMs. Training SVMs in distributed environment (e.g., MapReduce SVMs [30] and MPI SVMs [31]) is inefficient due to the iterative nature of the SVM training and costly network communication.

Platt [14] proposed binary SVMs with probabilistic output. Wu et al. [16] discussed different approaches to probability estimates for multi-class SVMs, and they have showed that the method based on pairwise coupling produces the best result. As a result, LibSVM implements pairwise coupling for MP-SVMs. A study [32] elaborated the advantages of one-against-all approach for classification. However, one-against-all is rarely used for probabilist SVMs in existing literature. Milgram et al. [33] compared several post-processing methods for MP-SVMs. They showed that SVMs estimate better probability than Multi Layer Perceptron (MLP). Although their techniques (e.g., softmax) can be used in GMP-SVM, we aim to produce the same SVMs as LibSVM. Using Milgram et al.'s techniques for post-processing is out of the scope of this paper.

Training SVMs using GPUs: Catanzaro et al. [12] first introduced GPUs for training binary SVMs. Wen et al. [18] proposed GPU based binary SVM cross-validation by pre-computing the whole kernel matrix which is stored in high-speed storage (e.g., SSDs). A recent study extended their algorithm for SVM regression problems [34]. Athanasopoulos et al. [35] used GPUs to purely accelerate the kernel matrix computation in the SVM training. These studies are for training binary SVMs and cannot handle large datasets, because the size of kernel matrix is quadratic in the number of instances. For example, processing the MNIST8M dataset with their algorithms needs 256TB of storage which is unacceptable for GPUs. Herrero-Lopez et al. [13] used one-against-all method to solve multi-class problems on GPUs. However, they represented the training instances in dense format for the ease of implementation and better memory alignment. The dense representation makes the above algorithms difficult to handle large but sparse datasets. Another study [23] compares different GPU SVM implements and provides some bench mark results, and proposes the **OHD-SVM** algorithm. However, the work only focuses on binary SVMs and no multi-class SVMs or probabilistic SVMs are presented. Cotter et al. [20] represented training instances in sparse format (i.e., CSR format [36]), and proposed a clustering technique to make use of the data sparseness. We also use CSR format to represent the training data for handling large but sparse datasets. We call Cotter et al.'s algorithm **GTSVM**. GTSVM supports both binary and multi-class SVMs, but does not support multi-class probability estimation due to their mathematical modelling. For a sanity check, we have compared GMP-SVM with GTSVM and OHD-SVM in Section 4.3. Our results have shown that GMP-SVM achieves the best performance among the various implementations.

6 Conclusion

In this paper, we have proposed a GPU based solution, called GMP-SVM, for multi-class SVMs with probabilistic output (MP-SVMs). The challenges of developing a highly parallelized GPU based solution for MP-SVMs are (i) repeated accesses to the high latency GPU memory, and (ii) the requirement of much larger memory footprint than the GPU memory. GMP-SVM reduces high latency memory accesses and memory consumption through batch processing, kernel value reusing and sharing, and support vector sharing. Experimental results have shown that GMP-SVM outperforms the GPU baseline by two to five times, and LibSVM with OpenMP by an order of magnitude. Additionally, GMP-SVM produces the same SVM model as LibSVM. With this significant performance speedup, we hope to bring MP-SVM a wider use in pattern recognition applications with increasing performance requirement.

Acknowledgements

This work is supported by a MoE AcRF Tier 1 grant (T1 251RES1610) and Tier 2 grant (MOE2017-T2-1-122) in Singapore. Prof. Chen is supported by the Guangdong special branch plans young talent with scientific and technological innovation (No. 2016TQ03X445), Guangzhou science and technology planning project (No. 2019-03-01-06-3002-0003) and Guangzhou Tianhe District science and technology planning project (No. 201702YH112). Bingsheng He and Jian Chen are corresponding authors. We thank NVIDIA for the hardware donations.

References

[1] L. Gruber, M. West *et al.*, "Gpu-accelerated bayesian learning and forecasting in simultaneous graphical dynamic linear models," *Bayesian Analysis*, vol. 11, no. 1, pp. 125–149, 2016.

[2] A. Nasridinov, Y. Lee, and Y.-H. Park, "Decision tree construction on gpu: ubiquitous parallel computing approach," *Computing*, vol. 96, no. 5, pp. 403–413, 2014.

[3] L. P. Jain, W. J. Scheirer, and T. E. Boult, "Multi-class open set recognition using probability of inclusion," in *European Conference on Computer Vision*. Springer, 2014, pp. 393–409.

[4] M. M. Rahman, B. C. Desai, and P. Bhattacharya, "Medical image retrieval with probabilistic multi-class support vector machine classifiers and adaptive similarity fusion," *Computerized Medical Imaging and Graphics*, vol. 32, no. 2, pp. 95–108, 2008.

[5] A. J. Joshi, F. Porikli, and N. Papanikolopoulos, "Multi-class active learning for image classification," in *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*. IEEE, 2009, pp. 2372–2379.

[6] C.-W. Hsu and C.-J. Lin, "A comparison of methods for multiclass Support Vector Machines," *IEEE Transactions on Neural Networks*, vol. 13, no. 2, pp. 415–425, 2002.

[7] C.-C. Chang and C.-J. Lin, "LIBSVM: A library for Support Vector Machines," *ACM Transactions on Intelligent Systems and Technology*, vol. 2, pp. 27:1–27:27, 2011.

[8] J. C. Platt, "Fast training of SVMs using Sequential Minimal Optimization," in *Advances in kernel methods*. MIT Press, 1999, pp. 185–208.

[9] M. A. Hearst, S. T. Dumais, E. Osman, J. Platt, and B. Scholkopf, "Support vector machines," *Intelligent Systems and their Applications*, vol. 13, no. 4, pp. 18–28, 1998.

[10] K. Q. Weinberger, F. Sha, and L. K. Saul, "Learning a kernel matrix for nonlinear dimensionality reduction," in *International Conference on Machine Learning*. ACM, 2004, p. 106.

[11] S. Shalev-Shwartz and T. Zhang, "Accelerated mini-batch stochastic dual coordinate ascent," in *Advances in Neural Information Processing Systems*, 2013, pp. 378–385.

[12] B. Catanzaro, N. Sundaram, and K. Keutzer, "Fast Support Vector Machine training and classification on graphics processors," in *Proceedings of the 25th International Conference on Machine learning*. ACM, 2008, pp. 104–111.

[13] S. Herrero-Lopez, J. R. Williams, and A. Sanchez, "Parallel multiclass classification using SVMs on GPUs," in *Workshop on General-Purpose Computation on Graphics Processing Units*. ACM, 2010, pp. 2–11.

[14] J. Platt, "Probabilistic outputs for Support Vector Machines and comparisons to regularized likelihood methods," *Advances in Large Margin Classifiers*, vol. 10, no. 3, pp. 61–74, 1999.

[15] H.-T. Lin, C.-J. Lin, and R. C. Weng, "A note on platt's probabilistic outputs for Support Vector Machines," *Machine learning*, vol. 68, no. 3, pp. 267–276, 2007.

[16] T.-F. Wu, C.-J. Lin, and R. C. Weng, "Probability estimates for multi-class classification by pairwise coupling," *Journal of Machine Learning Research*, vol. 5, no. Aug, pp. 975–1005, 2004.

[17] R.-E. Fan, P.-H. Chen, and C.-J. Lin, "Working set selection using second order information for training support vector machines," *Journal of machine learning research*, vol. 6, no. Dec, pp. 1889–1918, 2005.

[18] Z. Wen, R. Zhang, K. Ramamohanarao, J. Qi, and K. Taylor, "Mascot: fast and highly scalable SVM cross-validation using GPUs and SSDs," in *Data Mining (ICDM), 2014 IEEE International Conference on*, 2014, pp. 580–589.

[19] C. NVIDIA, "NVIDIA CUDA programming guide," 2011.

[20] A. Cotter, N. Srebro, and J. Keshet, "A GPU-tailored approach for training kernelized SVMs," in *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2011, pp. 805–813.

[21] T. Joachims, "Making large-scale svm learning practical," Technical Report, SFB 475: Komplexitätsreduktion in Multivariaten Datenstrukturen, Universität Dortmund, Tech. Rep., 1998.

[22] S. Tyree, J. R. Gardner, K. Q. Weinberger, K. Agrawal, and J. Tran, "Parallel support vector machines in practice," *arXiv preprint arXiv:1404.1066*, 2014.

[23] J. Vanek, J. Michalek, and J. Psutka, "A gpu-architecture optimized hierarchical decomposition algorithm for support vector machine training," *IEEE Transactions on Parallel and Distributed Systems*, 2017.

[24] K. Zhang, A. Guliani, S. Ogrenci-Memik, G. Memik, K. Yoshii, R. Sankaran, and P. Beckman, "Machine learning-based temperature prediction for runtime thermal management across system components," *IEEE Transactions on Parallel and Distributed Systems*, 2017.

[25] M. Grossman, M. Breternitz, and V. Sarkar, "Hadoopcl2: Motivating the design of a distributed, heterogeneous programming system with machine-learning applications," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 3, pp. 762–775, 2016.

[26] J. Yuan and S. Yu, "Privacy preserving back-propagation neural network learning made practical with cloud computing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 1, pp. 212–221, 2014.

[27] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, "The WEKA data mining software: an update," *ACM SIGKDD explorations Newsletter*, vol. 11, no. 1, pp. 10–18, 2009.

[28] T. Joachims, "Training linear SVMs in linear time," in *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2006, pp. 217–226.

[29] S. Shalev-Shwartz, Y. Singer, N. Srebro, and A. Cotter, "Pegasos: Primal estimated sub-gradient solver for SVM," *Mathematical Programming*, vol. 127, no. 1, pp. 3–30, 2011.

[30] F. O. Catak and M. E. Balaban, "CloudSVM: training an SVM classifier in cloud computing systems," in *Pervasive Computing and the Networked World*, 2013, pp. 57–68.

[31] L. J. Cao, S. S. Keerthi, C. J. Ong, J. Zhang, U. Periyathamby, X. J. Fu, and H. Lee, "Parallel Sequential Minimal Optimization for the training of Support Vector Machines," *IEEE Transactions on Neural Networks*, vol. 17, no. 4, pp. 1039–1049, 2006.

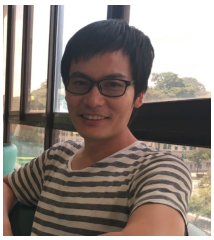
[32] R. Rifkin and A. Klautau, "In defense of one-vs-all classification," *Journal of machine learning research*, vol. 5, no. Jan, pp. 101–141, 2004.

[33] J. Milgram, M. Cheriet, and R. Sabourin, "Estimating accurate multi-class probabilities with Support Vector Machines," in *Proceedings of 2005 IEEE International Joint Conference on Neural Networks. (IJCNN)*, vol. 3, 2005, pp. 1906–1911.

[34] Z. Wen, R. Zhang, K. Ramamohanarao, and L. Yang, "Scalable and fast svm regression using modern hardware," *World Wide Web*, pp. 1–27, 2017.

[35] A. Athanasopoulos, A. Dimou, V. Mezaris, and I. Kompatsiaris, "GPU acceleration for Support Vector Machines," in *International Workshop on Image Analysis for Multimedia Interactive Services*, 2011.

[36] A. Buluç, J. T. Fineman, M. Frigo, J. R. Gilbert, and C. E. Leiserson, "Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks," in *Annual Symposium on Parallelism in Algorithms and Architectures*. ACM, 2009, pp. 233–244.



Zeyi Wen is a Research Fellow at National University of Singapore. Zeyi received his PhD degree in Computer Science from University of Melbourne in 2015, and a bachelor degree of Software Engineering from South China University of Technology in 2010. Zeyi's areas of research include high performance computing, machine learning and data mining.

Jiashuai Shi received his bachelor degree in Software Engineering from South China University of Technology in 2016. He is currently a master student at School of Software Engineering, South China University of Technology. His research interests include machine learning and high-performance computing.

Bingsheng He received the bachelor degree in computer science from Shanghai Jiao Tong University (1999-2003), and the PhD degree in computer science in Hong Kong University of Science and Technology (2003-2008). He is an Associate Professor in School of Computing of National University of Singapore. His research interests are high performance computing, distributed and parallel systems, and database systems.

Jian Chen is currently a Professor of the School of Software Engineering at South China University of Technology where she started as an Assistant Professor in 2005. She received her B.S. and Ph.D. degrees, both in Computer Science, from Sun Yat-Sen University, China, in 2000 and 2005 respectively. Her research interests can be summarized as developing effective and efficient data analysis techniques for complex data and the related applications.

Yawen Chen is currently a master student at School of Software Engineering, South China University of Technology, China. Her research interests include machine learning, and high-performance computing.